UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo de Fin de Grado

ANÁLISIS DE DATOS DE NETFLOW

Guillermo Hoyo Bravo
Javier Aracil Rico
Junio 2021

ANÁLISIS DE DATOS DE NETFLOW

AUTOR: Guillermo Hoyo Bravo
TUTOR: Javier Aracil Rico

Dpto. EPS, Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2021

Resumen

Este Trabajo de Fin de Grado se enfoca en el análisis de datos de netflow mediante diversas técnicas, de cara a analizar cuál de ellas es la más eficiente en base a unos criterios predeterminados. El análisis de los datos se lleva a cabo en el servicio web Grafana, el cual usa como motor de lectura de datos el servicio web Elasticsearch. Para subir los datos solicitados mediante Elasticsearch se ha usado el lenguaje Python junto con el servicio web Logstash. En Python procesamos los datos obtenidos en formato JSON y los almacenamos en CSV. Con estos nuevos ficheros, podemos subir a Elasticsearch el JSON desde un script de Python, mientras que los archivos CSV pueden enviarse desde la terminal de Linux con los comandos de Logstach. Todos estos servicios salvo Logstach se desplegaron desde Docker. Una vez subidos los datos, se hará uso de Grafana para crear sus respectivos Dashboards. Con ello podemos exponer los datos de muchas maneras diferentes según nuestros intereses. Este trabajo está más enfocado en la viabilidad de los procesos, se compara el tiempo que tarda cada proceso en ejecutarse completamente, tanto para pruebas como para los archivos reales, de igual manera que se compara el uso que le dan a los recursos, y la complejidad de realizar cada uno de ellos. Con la finalización de estos dos Dashboards, se comienza el proceso de experimentación, donde se buscará otra manera de optimizar los procesos o para su comparación. Este otro método consistirá en subir un archivo JSON desde Logstash para poder así cuanto influye la diferencia de formato para Logstash, la herramienta principal y más actual que merece más la pena ser estudiar.

Al final las diferencias principales son expuestas, las ventajas, las desventajas, así como los resultados obtenidos, y ciertas peculiaridades de cada proceso encontradas, por ejemplo, las diferentes maneras de medir el tiempo para cada forma. Habrá múltiples anexos con todos los códigos finales de programas utilizados para transformaciones de archivos, o iniciación de Web Services, como son los archivos YML o CONF. También se aporta la información necesaria para poder utilizar los programas mencionados. Hemos utilizado un programa secundario de visualización de datos, comprobando que el fichero de Elasticsearch contiene en todo momento los datos que introducimos originalmente, y que estos son correctos, pudiendo hacerse por comandos desde la terminal.

Un objetivo de este TFG es que cualquier persona que lo lea sea capaz de utilizar Docker, Grafana, Elasticsearch, Dejavu y Logstash. Sabiendo elegir el método más práctico para sus intereses o su caso. Este TFG es muy útil para el mundo laboral actual. Se exponen las diferentes situaciones donde cada método será más práctico, útil o posible de uso, para hacer más sencilla la síntesis de este trabajo. Además, durante el trabajo al completo, se van indicando diferentes motivos por el cual podemos fiarnos de los procesos que se han creado, desmontando la veracidad de ambos.

Palabras clave

Dashboard, Web Service, Netflow. Data Source, Elasticsearch, Plugins, Dejavu, Apache, Logstash, Kibana, Prometheus, PostgreSQL, Machine Learing, Data set, Bulk, parsear, Power BI.

Agradecimientos

Gracias a mi tutor Javier Aracil Rico por brindarme esta gran oportunidad de compartir un proyecto con él, de abrirme el proyecto personalmente pensando en mi CV y mis expectativas de futuro, por estar atento y hacerme sentir seguro, y por ayudarme con todo y de todas las maneras que ha podido. Siempre respetándose y siéndome muy cordial.

Gracias a Alejandro Peña Almansa, Graduado en la Universidad Autónoma de Madrid en Ingeniería de Telecomunicación, Máster en Ingeniería de Telecomunicación, y actualmente estudiante de PhD. en la UAM, por escucharme atentamente mejorando mi estado emocional y ayudarme con la comprensión lectora de mis expresiones.

Gracias a David Nevado: Licenciado en la Universidad autónoma de Madrid en Doble grado de Matemáticas e Informática y Licenciado en el Máster de Ciberseguridad: Por brindarme su apoyo moral

Por último, gracias a Raúl Torres Fernández: Estudiante de Ingeniería Informática de la Universidad Politécnica de Madrid y a Hugo Martins Ribeiro: Estudiante de Ingeniería Informática en la Universidad Complutense de Madrid, por ayudarme cuando más lo necesitaba y creía que no podía más o cuando me frustraba por el número de horas con un error típico y complicado de identificar en Python.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

La Memoria de este TFG está dirigida a estudiar distintas maneras de analizar datos de netflow, este tiene una gran importancia actualmente. Hoy en día, la mayoría de las empresas utilizan estas tecnologías. Son muy útiles para manejar grandes cantidades de datos y estudiarlas. También es un método muy útil de monitorizar el tráfico web y ver posibles erratas o errores.

1.1 Motivación

Los datos son la base de la informática. La informática está en constante cambio y empezó siendo un sitio perfecto y único para almacenar y gestionar datos. Esto, de manera colateral, ha hecho que otros sectores como el marketing, evolucionen, argumentado con la célebre frase "la Información es poder" [1]. La gestión de datos genera información, y el estudio de la información conforma el conocimiento.

Los datos podemos almacenarlos, modificarlos, consultarlos, enviarlos, codificarlos y decodificarlos de maneras muy sencillas. Al principio fue complicado, en 1950-1960 sólo sabíamos almacenar y leer datos [2] . En la década de los 70 se desarrollaron las Bases de Datos Relacionales (RDBSM). En los 80se crea el famoso lenguaje Structure Query Language (SQL), producto de IBM, y se investigan las bases de datos paralelas y distribuidas. En los 90 las bases de datos empiezan a desarrollarse tanto para realizar consultas como para la toma de decisiones, así como las bases de datos con interfaz web [3].

Poco a poco se fueron automatizando los procesos y creando herramientas para cada vez hacerlo de manera más cómoda y sencilla. Prácticamente toda la población mundial conoce el paquete de Microsoft Office (e.g. Excel o Word) , desarrollado en los 90 con el inicio de las Bases de Datos Orientadas a Objetos (OODBMS) [4], representando datos de manera vistosa e intuitiva, que facilitan nuestra comunicación y optimizan los recursos materiales, ya que solo consumen electricidad y los escasos materiales para construir servidores para almacenarlos, los cuales son inferiores a los gastos y problemas ambientales que producirían si los almacenaremos en papel, lo que antes se almacenaba en una biblioteca ahora sólo necesita unos pocos centímetros para ser accesible a todo el mundo.

Hoy en día, los datos se integran y utilizan de muchas más maneras. El análisis de datos está completamente introducido en la sociedad gracias a la automatización de almacenamiento de datos y gracias a la Inteligencia Artificial (IA). La IA es potenciada por los datos, poder acumular tanto es lo que permite que sean tan potentes, estas son las que nos llevarás a resolver los mayores problemas de la sociedad en el futuro.

La gestión de datos a gran escala está en las páginas webs, en el propio historial para personalizar tus anuncios, en los clics que hacemos. Está en los móviles, ordenadores, hasta en neveras y en coches, para saber a qué lugares viajas a menudo o dónde prefieres hacer la compra. Actualmente, existe tanta información y la tratamos de tantas maneras diferentes que los ordenadores nos conocen mejor que nosotros mismos.

Este es el motivo principal por el que es conveniente e inspirador el estudio de este preciso tema. Actualmente la IA está causando la cuarta revolución industrial [4], y cuando esta consiga ser tan compleja que se escape de nuestro entendimiento, empezará la quinta. Para que esto sea posible, los sistemas y programas deben ser prácticamente infalibles, con una precisión increíble. Esto lo conseguimos con ingentes cantidades de datos, mayormente. Otra

opción es mejorar los algoritmos del proceso de aprendizaje, lo que es más complicado, y evoluciona muy poco a poco, falta por descubrir del cerebro humano, y aun así no podemos confiar que sea algo remotamente posible. Por este motivo se optó por empezar desarrollando el método de procesamiento de datos masivos, en línea.

Los ámbitos que más se están desarrollando actualmente para lograr estos objetivos son: Big Data y análisis de datos, Machine Learning, Cloud Computing, Robótica y Ciberseguridad. Todo esto permite el manejo de datos a gran escala de manera más eficaz.

En concreto, este TFG se basa en el Big Data y el análisis de datos. Referencia [12] Tel uso de herramientas modernas como Docker, Elasticsearch y Grafana, son clave para obtener unos resultados impecables.

1.2 Objetivos

El objetivo de este TFG consiste en evaluar distintas maneras de analizar datos de netflow por la importancia y la demanda actual de estos conocimientos, para intentar mejorarlos en nuestra comunidad de informáticos y obtener mejores resultados.

El objetivo principal y obligatorio, es analizar el tráfico de red proporcionado en dos archivos DUMP realizando un Dashboard con la herramienta Grafana. Para esto disponemos de dos opciones principales.

- La primera opción que contemplamos es convertir los dos ficheros de datos DUMP a un único archivo JSON. Esto se realiza mediante Python, igual que la alimentación de un Elasticsearch usado desde Grafana para emplearlo como Datasource y crear el Dashboard donde analizar los datos.

- La segunda opción se basa en la conversión de los dos ficheros DUMP de datos a un único archivo CSV desde Python, alimentando un Elasticsearch con Logstash. De igual manera, este Elasticsearch será el motor para recoger los datos en Grafana y observarlos con el Dashboard

1.2.1 Tecnologías

Esta sección va a tratar las principales tecnologías utilizadas para llevar a cabo el TFG. Desde las más básicas para su realización, hasta otras para comprobar la correcta ejecución de los pasos intermedios, así como de guardado de archivos y datos (GIT).

Se incluyen también las definiciones de los distintos archivos y tipos de datos que se utilizan en cada tecnología, para profundizar más en ellos consultar los anexos.

Docker es una plataforma abierta[1] para el desarrollo, traslado y lanzamiento de aplicaciones. Te permite separar tu aplicación de tu infraestructura permitiendo una rápida entrega de software. Esto reduce drásticamente la preparación de un grupo de aplicaciones. Esta tecnología va a ser fundamental para montar el entorno de programación correcta y rápidamente. Para este TFG se crea un contenedor nuevo con las diferentes imágenes que

[1] https://www.docker.com/

necesitamos (Grafana, Elasticsearch, Dejavu) conectadas entre sí, aunque normalmente la mayoría ya han sido creadas por la comunidad.

Estas imágenes o contenedores prediseñados se pueden encontrar en el banco de imágenes oficial[2] y de la comunidad de Docker. Si encontramos aquí la imagen que es necesaria para realizar el trabajo, se ahorra una gran cantidad de tiempo y de carga de trabajo. De igual modo, la generación de un archivo Docker-compose.yml con las imágenes necesarias para nuestras necesidades, es más sencillo y rápido que la instalación por separado de cada programa en el equipo usado.

- **Docker-Compose** es la herramienta[3] necesaria para poder crear archivos donde ejecutar varias imágenes a la vez en un mismo contenedor, pudiendo modificar diferentes valores, como la versión que descargar del programa, el puerto de acceso y otros valores muy relevantes. Con Docker-Compose, el programador es capaz de personalizar las aplicaciones, conexiones y puertos necesarios para el proyecto y un entorno únicos.

  Vamos a instalar Docker y Docker-Compose. Para ello seguimos los pasos que hay en la página oficial. Explicaremos este proceso en el capítulo de desarrollo.

  - **Docker-compose.yml**: (YAML Ain't Markup Language) es un lenguaje de serialización de datos legible[4] por humanos. Por lo general, se usa para archivos de configuración y aplicaciones que se usan para almacenar o transferir datos. En Docker, se utiliza para instalar diferentes imágenes de servicios web o programas para usarlos juntos y definir su configuración. Aquí se escriben datos como la IP que vas a visitar, el puerto utilizado para realizar esta operación o la conexión entre estos programas, y luego todo el entorno es muy fácil de configurar.

Grafana es un software libre[5], basado en Apache 2.0, para la visualización y formato de datos de serie temporales a partir de su recolección. Es multiplataforma, sin dependencias y con la posibilidad de implementación con Docker.

Hoy en día es la tecnología más popular para el desarrollo de gráficos y cuadros de mando / Dashboard. Contiene una muy inmensa cantidad de plugins (extensiones de funcionalidad) a instalar para realizar el trabajo de manera más sencilla, personalizada y detallada. Esta sección es desarrollada mayormente por la comunidad[6]. Ejemplos de estos plugins son Prometheus, Elasticsearch, PostgreSQL y JSON API

[2] https://hub.docker.com/

[3] https://docs.docker.com/compose/

[4] https://en.wikipedia.org/wiki/YAML

[5] https://grafana.com

[6] https://grafana.com/grafana/plugins/

Para poder empezar a progresar con los desarrollos, además de conocimientos de Python y de Elasticsearch, que se investigarán más adelante, vamos a empezar por aprender de Grafana con el tutorial básico oficial[7].

Con Grafana en este TFG, se observan datos de diferentes archivos convertidos desde los archivos originales suministrados de tipo DUMP. Los nuevos archivos son subidos a Elasticsearch para poder contemplarlos con Grafana. Una breve introducción a estos archivos:

- **JSON**: (JavaScript Object Notation) [6] Archivo de formato de intercambio de datos muy liviano, sencillo de leer y veloz también para escribir para humanos, así como de convertir y generar para ordenadores.

- **JSONL**: (JSON Lines8) Gran archivo para logs. Es flexible para compartir mensajes entre procesos que cooperan. Guarda datos de manera estructurada para su procesamiento uno a uno.

- **CSV**: (Comma-Separated Value) [7] Este formato de archivo representa un array de valores numéricos y de texto. Es un ejemplo de "archivo plano". Este archivo delimita sus datos de columnas por comas.

- **DUMP**: Resultado del volcado de la memoria de una base de datos [8]. Este archivo no tiene una estructura definida y consta de los datos hasta el preciso momento de su creación.

Elasticsearch es conocido por sus capacidades de indexación[9] de múltiples tipos de contenido. Posteriormente, esto tiene una gran variedad de usos, desde la búsqueda de aplicaciones o en sitios webs, así como mostrar su rendimiento, generar analíticas de seguridad, negocios y de datos.

Logstash uno de los productos principales de Elasticsearch. Se usa para agregar, procesar y enviar datos a Elasticsearch. Logstash[10] es un pipeline de procesamiento de datos open Source y del lado del servidor, que te permite ingresar datos de múltiples fuentes simultáneamente, así como enriquecerlos y transformarlos antes de que se indexen en Elasticsearch.

- **Logstash.conf**: Es el archivo encargado de configurar banderas de Logstash [9]. La entrada de los datos será definida aquí, igual que el formato que trae la entrada de datos y la salida, en nuestro caso esta última será el servicio web de Elasticsearch.

[7] https://grafana.com/tutorials/

[8] https://jsonlines.org

[9] https://www.elastic.co/es/

[10] https://www.elastic.co/es/logstash

Dejavu es un servicio web para visualizar[11] la información de un JSON o un CSV subido a Elasticsearch. Elasticsearch carece de un modo completamente desarrollado para ver la información de sus índices. Se puede ver desde la terminal, de manera muy limitada y caótica, por lo que, aunque mostremos ambas opciones más adelante, Dejavu nos ayudará mucho a tener un desarrollo más limpio y vistoso.

Estos programas son ejecutados desde Docker, excepto el de Logstash.

1.2.2 Conocimientos

Los conocimientos necesarios para este TFG son básicos de Python, de manejo de ficheros y de datos. Es necesario entender los diferentes archivos de datos para poder convertirlos. Es necesario el uso de Linux y los conocimientos básicos de la terminal para descargar e instalar los programas necesarios, así como para ir a los directorios que sea preciso para ejecutar un programa como Logstash.

Se requieres conocimientos de Docker mayoritariamente básicos, así como unos algo más avanzados de Docker-Compose. Es fundamental conocer los procesos de Elasticsearch, sus comandos y funcionamiento, así como conocimientos extras como su método bulk para poder subir datos directamente desde Python. Los conocimientos básicos de Logstash también son muy importantes, así como comandos de ejecución y como generar un logstash.conf apropiado para el trabajo.

Por último, conocimientos de Grafana, como añadir Datasource, cómo configurarlos, así como saber cómo crear Dashboards y filtrar el contenido. Aparte de esto, se añade el conocimiento de otra tecnología auxiliar llamada Dejavu para poder controlar de manera más visual y sencilla los datos subidos a Elasticsearch.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Capítulo 1: Introducción. En este capítulo se presentan los siguientes apartados: la motivación, los objetivos, las tecnologías utilizadas, los conocimientos finales obtenidos, conjunto a la organización del trabajo.

- Capítulo 2: Estado de Arte. En este capítulo se realiza un estudio de las tecnologías que se utilizan, y de los diferentes usos o resultados que ya se han obtenido con estas. Para este TFG en concreto se ha buscado información acerca del uso conjunto de Elasticsearch con Grafana.

- Capítulo 3: Desarrollo. En este capítulo se desarrolla formalmente los procesos de desarrollo del trabajo. Se explica detalladamente los pasos seguidos para poder realizar la subida de los diferentes archivos datos desde Python y desde Logstash a Elasticsearch y como generar los diferentes Dashboards.

[11] https://opensource.appbase.io/dejavu/

- Capítulo 4: Integración Pruebas y Resultados. A lo largo de este capítulo se desarrollan los diferentes experimentos realizados, y las opciones alternativas que se han estudiado.

- Capítulo 5: Conclusión y Trabajo Futuro. Este apartado recopila las principales conclusiones del trabajo, sintetizando el trabajo para el lector, y encaminando los futuros pasos que podría seguir este trabajo como investigación.

2 Estado del arte

En este capítulo se recopila la investigación realizada para analizar las diferentes tecnologías y trabajos que han sido necesarias para este TFG. También se ha comprobado si es un trabajo de ya investigado, o en su defecto, como es el caso, cuanta información similar existe y hasta qué punto son comparables.

2.1 Docker para montar el entorno

Existe relativamente escasa información referente a conectar Grafana con Elasticsearch. En la mayoría de información disponible, se explica de manera muy superficial, normalmente se pone el foco en la parte final, donde se explica cómo seleccionar un Datasource previamente creado con Elasticsearch desde la creación del Dashboard [10]. Explican consultas específicas para el Data set especifico, y como mostrar o filtrar los diferentes parámetros, de forma más general. De manera similar a lo que explican en la documentación de Grafana y de los tutoriales hablados anteriormente.

Gracias a un tutorial, donde se crean un Docker-compose.yml[12] en el que conecta los tres servicios mencionados anteriormente, haciendo que se puedan utilizar de manera muy sencilla. Desde entonces Docker pasó a ser la opción principal para este trabajo, ya que la instalación de estas tecnologías fundamentales se hace algo pesada. También aparece otra manera de subir los datos a Grafana desde Elasticsearch que es una fuente de investigación y experimentación, pudiendo compararse con las dos anteriores, para poder escoger la mejor de las tres.

Esto fue un gran punto a favor, por el hecho de que Elasticsearch es muy usado por otro visualizador y gestor de datos llamado Kibana. Lo más frecuente es encontrar información acerca de la pila ELK, parejo a la tecnología de Grafana el dataset de Prometheus [11]. Otra información encontrada para conectar Grafana Elasticsearch y Docker reside en páginas profesionales como LinkedIn, donde existen amplios conocimientos de Docker y Docker-Compose [12], haciendo más fácil, viable y moderno este trabajo.

2.1.1 Subir datos a Elasticsearch

La complejidad de subir archivos JSON a Elasticsearch mediante Python reside en la costosa documentación de sus métodos, al ser una funcionalidad algo relativamente nueva. Por su implementación resulto más larga que las demás. Existen páginas con métodos obsoletos, y sin amplios conocimientos de conexión de Python a Web Services que puede resultar engorroso. Haciendo uso de un fichero de ejemplo suministrado para este trabajo y la efusiva investigación[13] en Internet acabo resultando un trabajo más sencillo.

[12] https://www.metricfire.com/blog/using-grafana-with-elasticsearch-tutorial/

[13] https://kb.objectrocket.com/elasticsearch/how-to-bulk-index-elasticsearch-documents-from-a-json-file-using-python-753

2.1.2 Procesar Información en Grafana

De igual modo, cuesta encontrar información sobre cómo funciona Grafana para conectarse a Elasticsearch y recoger los datos por columnas, para explicar cómo tiene que ser el formato exacto para que Grafana identifique cada dato.

De igual modo, cada gráfico es único, y hay muchos tutoriales de cómo realizar distintos datos una vez la información ya está subida al Datasource de Grafana. Ejemplos de esto sería el tutorial comentado anteriormente con el que aprendemos a usar Grafana, y el video de YouTube también mencionado con anterioridad.

2.1.3 Conversión de Archivos

Este tema es muy conocido y utilizado mundialmente. Los archivos de datos están precisamente para usarlos. Se habla de tipos de archivos universales como son DUMP y JSON, por lo que es muy fácil acceder a información de la conversión entre estos. Igualmente, el modo de trabajo empleado ha sido informarse acerca del formato de los diferentes archivos funcionales para Elasticsearch y Grafana, y trabajar con este conocimiento. Una vez realizado, si aparecía algún error en el entorno utilizado (PyCharm), se pasaba a buscar ejemplos fotográficos, para poder identificar los detalles que pudieran faltar (ver Figura 2.1).

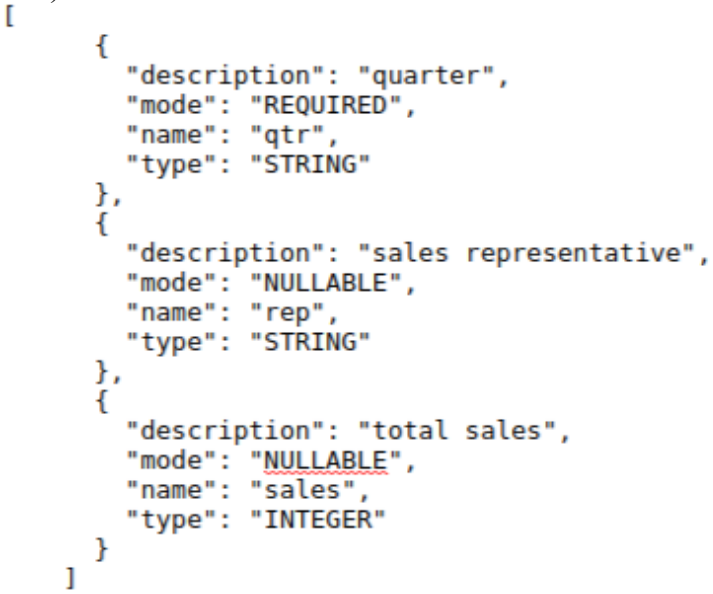


Figura 2.1: Imagen del formato de archivo JSON

En resumen, la información puede ser algo compleja de encontrar, hay que ir poco a poco, y no se ha conseguido encontrar estudios o información completa acerca de los objetivos obligatorios planteados. Es posible encontrar información dispersa en distintos foros, pero no es tan sencillo como juntarlos, porque esto genera errores y resultados que no son óptimos.

3 Desarrollo

Este apartado explica el proceso completo realizado en este TFG para lograr los objetivos planteados. Para ello, se explica los pasos que se han realizado desde el principio hasta el final.

3.1 Preparación de entorno

En este apartado se explica la instalación completa de las herramientas y tecnologías que han sido necesarias para la realización del TFG.

3.1.1 Prerrequisitos

Hay prerrequisitos imprescindibles para el tutorial de Grafana oficial. Estos son Docker, Docker-Compose y Git. En este caso usaremos Git para ir guardando los archivos de configuración, los diferentes programas de Python, así como ficheros de datos generados. Usaremos Docker y Docker-Compose para instalarnos los programas necesarios para realizar el trabajo. Pasamos a exponer la instalación de cada tecnología necesaria.

Docker

Para la instalación de Docker, es muy útil seguir los comandos de la página oficial[14]. Hay varias maneras de hacerlo, y en concreto, se va a instalar usando un script proporcionado.

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

Es recomendable usar este comando para poder usar Docker sin ser usuario root. Necesita reinicio:

```
$ sudo usermod -aG docker <your-user>
```

Probar que está correctamente instalado:

```
$ docker -v
OUTPUT
[USER@]:~$ docker -v
Docker version 20.1.3, build 48d30b5
```

Comprobar que Docker se está ejecutando:

```
$ docker ps
OUTPUT
[USER@]:~$ docker ps
CONTAINER ID IMAGE COMMAND STATUS PORTS NAME
```

Compose es una herramienta para definir y ejecutar contenedores de Docker de varios contenedores. Compose, usa un archivo YAML (Docker file) para configurar los servicios de su aplicación. Luego, con un solo comando, crea e inicia todos los servicios desde su configuración.

[14] https://docs.docker.com/engine/install/ubuntu/

Se va a instalar Docker Compose de manera similar a la anterior, utilizando los comandos de la página oficial.[15]
Con este comando se descarga la versión más actual de Docker-Compose:

```
$ sudo curl -L "https://github.com/docker/compose/releases/download/1.28.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Se aplican los permisos necesarios para el archivo binario:

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

Probar que la instalación funciona:

```
$ docker-compose –versión
```

OUTPUT

```
[USER@]:~$ docker-compose --version
docker-compose version 1.28.2, build 67630359
```

Git

Ahora se va a instalar Git, para ello vamos a la página oficial, recomendada en la tutoría, este proceso es muy sencillo[16], por lo que no se explica.

Probar que la instalación funciona:

```
$ git --version
```

OUTPUT

```
[USER@]:~$ git --version
git version 2.30.0
```

Java

Para Java tampoco se explica el proceso en este trabajo, suponemos que son las tecnologías más básicas y que se conocen más, incluso puede que Java venga instalado. Simplemente se proporciona el enlace de donde se han sacado los comandos precisos para instalarlo.[17]

Para comprobar la instalación:

```
$ java --version
```

OUTPUT

```
openjdk version "11.0.7" 2020-04-14
OpenJDK Runtime Environment (build 11.0.7+10-post-Ubuntu-3ubuntu1)
OpenJDK 64-Bit Server VM (build 11.0.7+10-post-Ubuntu-3ubuntu1, mixed mode, sharing)
```

[15] https://docs.docker.com/compose/install/

[16] https://git-scm.com/download/linux

[17] https://www.itzgeek.com/post/how-to-install-elk-stack-on-ubuntu-20-04/

3.1.2 Instalación

En este apartado se va a explicar cómo se han instalado las herramientas necesarias para el entorno. Se seguirán comandos y se referenciarán las respectivas páginas oficiales[18] necesarias. Se descarga la clave pública.

```
$ wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
```

Descargamos el paquete necesario

```
$ sudo apt-get install apt-transport-https
```

Guardamos el repositorio

```
$ echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-7.x.list
```

```
$ sudo apt-get update && sudo apt-get install logstash
```

Seguidamente se prepara el entorno entero donde se trabaja de forma que esté operativo desde el primer momento. Esto se hace ejecutando el archivo Docker-compose.yml. Para ello, se utilizan los siguientes 3 puertos en localhost:

- Url Elasticsearch: http://localhost:9200/
- Url Dejavu: http://localhost:1358/
- Url Grafana: http://localhost:3000/

Estas URL son el salvoconducto para saber si los servicios están operativos correctamente. Los comandos de servicio son muy útiles también para poder comprobar el estado de los servicios si alguno falla, al ser comandos del manual del programador:

```
$ sudo systemctl status <webservice>
```

Salida Terminal después del Stop o si el servicio está fallando.

```
● logstash.service - logstash
   Loaded: loaded (/etc/systemd/system/logstash.service; disabled; vendor pre>
   Active: failed (Result: exit-code) since Mon 2021-05-17 17:28:31 CEST; 16s>
  Process: 18712 ExecStart=/usr/share/logstash/bin/logstash --path.settings />
 Main PID: 18712 (code=exited, status=1/FAILURE)
…
```

[18] https://www.elastic.co/guide/en/logstash/current/installing-logstash.html

Salida Terminal después de Start o si el servicio este operativo.

```
● logstash.service - logstash
   Loaded: loaded (/etc/systemd/system/logstash.service; disabled; vendor
        pre>
   Active: active (running) since Mon 2021-05-17 17:30:23 CEST; 5s ago
 Main PID: 18984 (java)
    Tasks: 28 (limit: 9336)
   Memory: 470.0M
   CGroup: /system.slice/logstash.service
           └─18984 /usr/share/logstash/jdk/bin/java -Xms1g -Xmx1g -
           XX:+UseCon>
…
```

Los comandos más básicos necesarios para modificar el estado de estos servicios[19] son:

$ sudo systemctl start <webservice>

$ sudo systemctl stop <webservice>

Es una manera muy útil de verificar que han sido instalados correctamente, si están funcionando actualmente, o para pararlo si es lo que queremos. Funciona tanto para Docker como para los 3 web services recién mencionados, como para Logstash.

Lo primero y fundamental para ejecutar estos servicios webs es definirlos en un Docker-compose.yml, para su ejecución automática en contenedores, con sus respectivas conexiones. Para ver este archivo ir al **Anexo C.C Docker-Compose.yml**

Una vez este el archivo preparado para su ejecución es necesario introducir este comando en la terminal:

$ docker-compose up -d

Este comando debe ejecutarse desde la carpeta donde se encuentre el archivo docker-compose.yml

Comprobamos que la salida de los contenedores es correcta:

```
: ~$ docker-compose up -d
Building with native build. Learn about native build in Compose here:
        https://docs.docker.com/go/compose-native-build/
Starting elasticsearch ... done
Starting tfg_grafana_1  ... done
Starting tfg_logstash_1 ... done
Starting dejavu         ... done
```

Aquí es donde entran en juego las 3 URL que hemos mencionado antes. Al acceder a cada una de ellas, si se están ejecutando correctamente se debería poder ver la información mostrada en las Figuras 3.1, 3.2 y 3.3.

[19] https://www.digitalocean.com/community/tutorials/how-to-use-systemctl-to-manage-systemd-services-and-units

```
{
  "name" : "5743ccb234d5",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "G07X6_r3TSy3ufvRL7oQTA",
  "version" : {
    "number" : "7.0.1",
    "build_flavor" : "oss",
    "build_type" : "docker",
    "build_hash" : "e4efcb5",
    "build_date" : "2019-04-29T12:56:03.145736Z",
    "build_snapshot" : false,
    "lucene_version" : "8.0.0",
    "minimum_wire_compatibility_version" : "6.7.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```


Figura 3.1. Funcionamiento correcto del web Service Elasticsearch. Donde se encuentra la información básica de este.

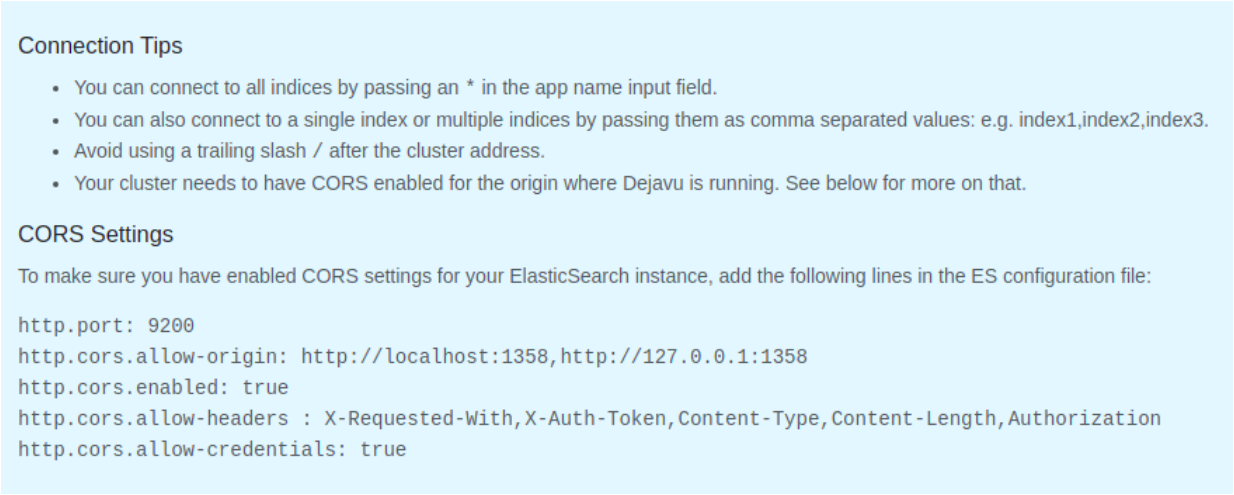


Figura 3.2. Funcionamiento correcto del webservice Dejavu. Se muestra la página principal con los tips de conexión y algunas opciones.

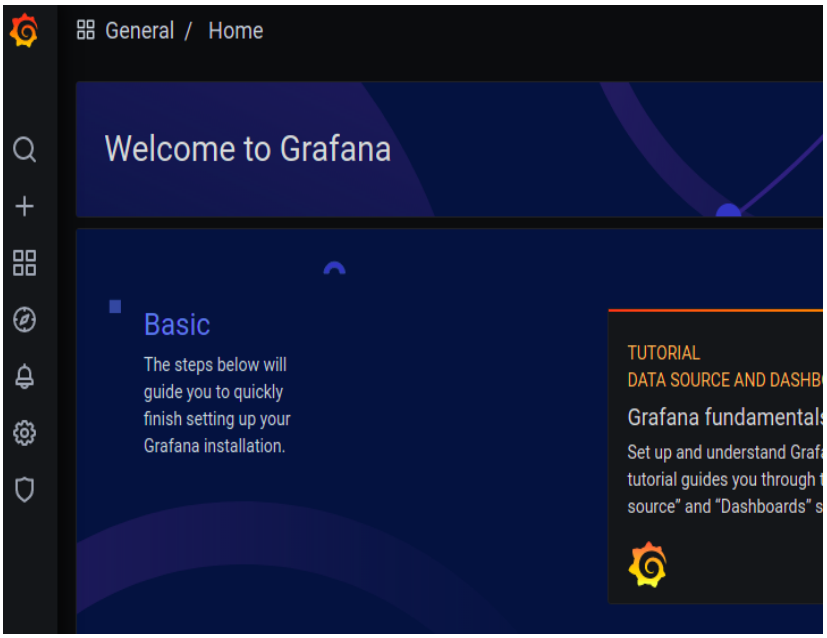


Figura 3.3 Funcionamiento correcto del web Service Grafana. Se muestra la página principal una vez iniciada sesión, con el menú de opciones a la izquierda. Inicio de sesión con usuario y contraseña: admin, admin.

3.2 Conversión de Archivos

Este apartado se centra en las distintas conversiones de archivos y en cómo se han realizado, el código de cada programa se adjuntará como anexos.

3.2.1 JSON

El formato JSON es conocido mundialmente, y ya había trabajado antes con él. Por este motivo decidí empezar por aquí para subir la información a Grafana. Transformar el DUMP a JSON resultó sencillo, simplemente tenía que recopilar la información de los distintos archivos en un diccionario con el nombre de cada columna, y volcar cada fila del diccionario en un nuevo archivo, JSON. Al principio encontré un par de dificultades con el formato, investigando un poco, rápidamente pude conseguir terminar el archivo de modo correcto.

Este formato empieza y termina con corchetes. Entre medias, encontramos los datos en formato: < “Nombre del campo”: Dato>, separados por comas y entre llaves, de forma que cuanto termina la fila de datos, se cierra la llave y se pone una coma para introducir que viene uno nuevo.

Ejemplo:
[{“Columna1”: “Dato1”, “Columna2”: “Dato2”, “Columna 3”: 3},
{“Columna1”: “Hola”, “Columna2”: “Aloha”, “Columna 3”: 8},
{“Columna1”: “Adiós”, “Columna2”: “GoodBye”, “Columna 3”: 10}]

Esto no era suficiente, la forma de subir datos a Elasticsearch no es la de un JSON habitual.

3.2.2 JSON Elasticsearch

El formato JSON que necesitaba era muy similar al normal, quitando el corchete inicial y final que indican y el inicio y el final del archivo, así como las comas de final de línea de datos en el JSON.

Solamente era necesario meter las líneas de datos en formato JSON separadas por un salto de línea. Por lo que pasar del programa que realiza de pasar el DUMP a JSON me resultó muy fácil de modificar para crear este nuevo programa.

El formato queda de la siguiente manera:
{“Columna1”: “Dato1”, “Columna2”: “Dato2”, “Columna 3”: 3},
{“Columna1”: “Hola”, “Columna2”: “Aloha”, “Columna 3”: 8},
{“Columna1”: “Adiós”, “Columna2”: “GoodBye”, “Columna 3”: 10}º

Se adjunta programa en **Anexo A.2**

3.2.3 CSV

El formato CSV también consiste en un programa similar a los dos de conversión a JSON. En este caso, la estructura de un CSV es: una fila con los nombres de cada columnas separadas por comas, y las siguientes filas con los datos correspondiente a cada columna separado por comas. Ejemplo:

Columna1	Columna2	Columna 3
Dato 1,	Dato 2,	3
Hola,	Aloha,	8
Adiós,	Goodbye,	10

Tabla 3.1 Tabla ejemplificativa de un archivo CSV

Se adjunta programa completo de conversión de los archivos DUMP a CSV en el Anexo BB.

3.3 Subir Archivos a Elasticsearch

Este apartado trata las diferentes maneras de subir el contenido de los ficheros a Elasticsearch para después poder visualizarlo. Se comprueba mayoritariamente lo que tarda cada uno de ellos y el proceso que realizan para poder valorar otra característica de las opciones a evaluar. Para poder trabajar con Elasticsearch desde Python, es necesario descargarse[21] un cliente para que se puedan coordinar en la misma área de trabajo.

Para cada apartado de este proceso, para cada manera de subir los datos, usamos los ficheros de prueba encontrados en el Anexo A (A.A y A.B).

3.3.1 Subir Archivos por medio de Python

Para subir los archivos de un JSON desde Python, se ha creado un programa que se conecta con el cliente de Elasticsearch de esta manera:

```
cliente = Elasticsearch(“localhost:9200”)
```

[21] https://elasticsearch-py.readthedocs.io/en/7.10.0/

El bulk de datos se realiza mediante la función helpers.bulk("Cliente Elastic", "Elasticsearch Doc", "Nombre Indice", "Archivo con Datos"): De esta manera:

Respuesta = helpers.bulk(cliente, doc_list, "tfg-json-elastic", doc_type = "TestElasticsearch.json")

Para ver toda la información relevante a este programa consultar el Anexo C.C.

Esta manera es bastante útil para subir los datos. Además, es eficaz, sencilla de ejecutar, es simplemente un fichero de Python con una referencia al fichero JSON que deseamos subir a Elasticsearch. Se ejecuta como cualquier otro programa Python:

$ python3 "nombre del programa".py
$ python3 UploadToElasticsearch.py

Este programa contiene una línea de código para medir el tiempo de ejecución y saber cuánto tardan en subirse los archivos a Elasticsearch.

Este proceso resultó complejo, una vez comprendido el modo de funcionamiento de la librería helpers y la función bulk de Elasticsearch, el proceso se vuelve más fácil. Esto tardo más de lo esperado, se probaron diversas maneras para conseguir el funcionamiento correcto de la librería hasta dar con la adecuada.

3.3.2 Subir Archivos por medio de Logstash

Para subir los datos con Logstash, además de tenerlo descargado y operativo. es necesario que lo ejecutemos desde la carpeta donde está instalado, para esto puedes usar el comando:

$ sudo whereis logstash.
Respuesta: logstash: /etc/logstash /usr/share/logstash

En este caso, aparecen dos direcciones, la correcta es donde se encuentra la carpeta/bin, esta carpeta es: /usr/share/logstash/bin.

$ sudo cd /usr/share/logstash/bin
Dirección Actual: /usr/share/logstash/bin

Desde aquí sí se permite ejecutar comandos de Logstash, a diferencia de en cualquier otro directorio. Para esto, y para comprobar que Logstash está ejecutándose correctamente, primero voy a ejecutar el comando:

$ sudo ./logstash –version.

Ahora que Logstash se está ejecutando, se crea un archivo´ CONF para Logstash, en esta carpeta, llamado logstash.conf. Se adjunta el código en el **Anexo B.2**. Una breve explicación de este archivo es que consta de tres partes:

- Input: es el archivo CSV. Esta parte es la encargada de recibir los datos de entrada.
- Filter: es el proceso intermedio. La interpretación, el formato y procesamiento de los datos de entrada.
- Output: es la salida de los datos. En nuestro caso la conexión y el índice de Elasticsearch.

Ahora, se ejecuta el comando:

$ sudo ./logstash -f logstash.conf

Desde la ubicación mencionada: */usr/share/logstash/bin*. Ejecutamos el archivo logstash.conf que hemos creado, leyendo, procesando y subiendo los datos a Elasticsearch.

Se observa la información se va mostrando en la terminal mientras se sube cada dato/columna de datos. Lo que aporta claridad y veracidad de su correcto funcionamiento, haciendo que el usuario se sienta más seguro a la hora de trabajar con este programa.

Aquí se mide el tiempo desde que se ejecuta el comando hasta que termina la respuesta por la terminal.

$ sudo time ./logstash -f logstash.conf

Este proceso ha sido muy sencillo, lo más complicado ha sido ejecutar correctamente el comando y en la ubicación precisa. Es fácil de entender y bastante visual, me ha sorprendido este programa tanto en eficacia como en facilidad de uso para el usuario.

3.4 Visualización de Datos

Una vez que los datos se encuentran subidos o indexados en Elasticsearch, se puede comprobar que el proceso es correcto, para otorgarle veracidad al trabajo. Así también se consigue que los lectores de este puedan fiarse y se atrevan a probarlo por ellos mismo. Hay varias maneras para hacer esto:

3.4.1 Desde Terminal

Como se ha mencionado anteriormente, cuando ejecutamos el comando de Logstash, se observan directamente los datos que se procesan según se suben. Esta forma de ver los datos sólo valdría para Logstash, además de que los datos pasan muy rápidos dándonos poco tiempo para pararnos a verlos, hasta el final, donde si hay muchos datos es poco práctico, y buscarlos también se puede complicar. Otra manera de ver los datos por terminar es usando los siguientes comandos de Elasticsearch:

A continuación, se observa[23] comando con el que visualizamos los índices, y la tabla 3.2 con los resultados de este comando.

$ curl -XGET localhost:9200/_cat/indices?v

health	status	index	uuid	pri	rep	docs.count	docs.delete	store.size	pri.store.size
yellow	open	tfg-json-elastic	CHYympPKQpuMSXOqC9y0mw	1	1	8	0	11.4kb	11.4kb
yellow	open	tfg-csv-logstash	fWvFKUY0SRuYwPOdn2C4Gw	1	1	8	0	88.1kb	88.1kb

Tabla 3.2. Tabla mostrando los resultados de la búsqueda de incides de Elasticsearch por terminal.

[23] https://www.bmc.com/blogs/elasticsearch-commands/

Ahora el comando de visualización de datos en un índice:

$ curl -X GET http://localhost:9200/NombreIndiceABuscar/_search
$ curl -X GET http://localhost:9200/tfg-json-elastic/_search

Respuesta de terminal: muestra una fila entera del fichero de datos subido.

{"took":53,"timed_out":false,"_shards":{"total":1,"successful":1,"skipped":0,"failed":0},"hits":{"total":{"value":8,"relation":"eq"},"max_score":1.0,"hits":[{"_index":"tfg-json-elastic","_type":"TestElasticsearch.json","_id":"saT0x3kBF4qGo8Aa_II4","_score":1.0,"_source":{"timestamp-ts1":"2020-09-08T16:15:00-05:00","timestamp-ts2":"2020-09-08T16:15:00-05:00","timestamp-ts3":"2020-09-08T16:15:00-05:00","pr":6,"ips-ra":"192.168.29.3","ips-sa":"201.116.65.2","ips-da":"201.116.168.245","ports-sp":30758,"ports-dp":443,"package_action-iin":0,"package_action-out":0,"package_dir-ipkt":9,"package_dir-opkt":0,"bytes_dir-ibyt":2855,"bytes_dir-obyt":0,"fl":1,"rates-bps":7633,"rates-pps":3,"rates-bpp":317,"timestamp":"2021-06-01T16:23:11.816782"}},{"_index": …

Esta es la salida de un único dato que se muestra por terminal, el segundo elemento empezaría de nuevo indicando el índice en el que se encuentra ({"_index": …). A diferencia de la monitorización con Dejavu, que es más sencilla para el programador, por eso más adelante explicaremos este webservice.

Si queremos eliminar un índice, el comando indicado es:

$ curl -XDELETE localhost:9200/search
$ curl -XDELETE localhost:9200/tfg-csv-logstash

Resultado:

{"acknowledged":true}

Resultado mostrado por terminal de eliminar todos los índices y buscar índices subidos:

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

De esta manera se ve todo lo que hemos subido, comparando el primer y el último dato se comprueba deberían ser correctos. De igual manera, este modo es confuso y poco limpio además de impreciso, por lo que, aunque está bien y es sencillo, vamos a ayudarnos de Dejavu.

3.4.2 Con Dejavu

La configuración realizada anteriormente de Docker-compose.yml conecta Dejavu con Elasticsearch para poder recoger los datos, por lo que se puede entrar a la URL de Dejavú. Seleccionar en el campo de URL donde introducir la de Elasticsearch, y en el campo de índex el escribir el respectivo índice que queremos visualizar, al pulsar en buscar se reciben todos los datos ordenados, y con divisiones por sus respectivos campos o columnas.

Para esto, en el campo superior izquierdo, URL, colocaremos la dirección de la API de los datos, es decir: http://elasticsearch:9200

En el campo superior derecho, colocamos el índice correspondiente del que queremos ver los datos: tfg-json-elasticsearch. Esto se observa perfectamente en la siguiente Figura 3.4.

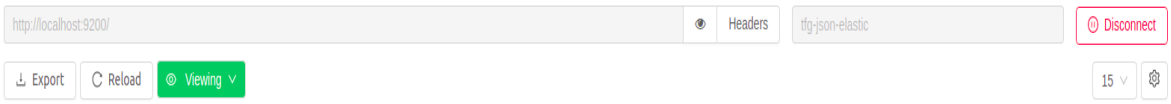


Figura 3.4. Se muestra cómo se contemplan los datos de un índice de Elasticsearch, conectándonos a este desde Dejavu. Se puede apreciar la conexión a Elasticsearch (http://localhost:9200), así como la conexión al índice deseado (tfg-json-elastic)

{…}_id	bytes_dir-ibyt	bytes_dir-obyt	fl	ips-da	ips-ra	ips-sa	package_act	…
01{…} uKT0x3kBF4qGo8Aa_114	64	0	1	172.22.201.80	172.22.201.34	172.22.2.6	219	…
02 {…} tqT0x3kBF4qGo8Aa_114	164	0	3	172.16.215.8	172.16.201.213	172.16.216.123	868	…
…	…	…	…	…	…	…	…	…

Tabla 3.3. Pequeña tabla mostrando la información de un índice como lo representa Dejavu. Cada columna es un dato de cada línea del fichero. Cada fila representada es una línea completa. Se puede observar que la tabla está incompleta tanto en filas como en columnas porque esto ocuparía mucho más espacio del necesario para una representación de los datos.

3.4.3 Análisis de datos con Grafana

Para analizar los datos con Grafana, lo prioritario es acceder al servicio web, con usuario/contraseña: admin/admin. Una vez dentro, vamos a añadir los Datasource correspondientes. Como hemos subido los datos de distintas maneras, pero todos a Elasticsearch con diferentes índices, vamos a añadir tantos Datasource de Elasticsearch como índices tenemos, con diferentes nombres para su distinción. Observar Figura 3.4 (a) y 3.4 (b). Este es un proceso es genérico para añadir Datasource y observar los resultados.

3.4.4 Análisis de datos JSON

Introducción del índice en Elasticsearch deseado como Datasource, para la creación de un nuevo Dashboard. En la creación de este Dashboard, seleccionamos el Elasticsearch con los datos deseados, y seleccionamos los parámetros o campos por los que queremos realizar el filtrado y muestreo de datos.

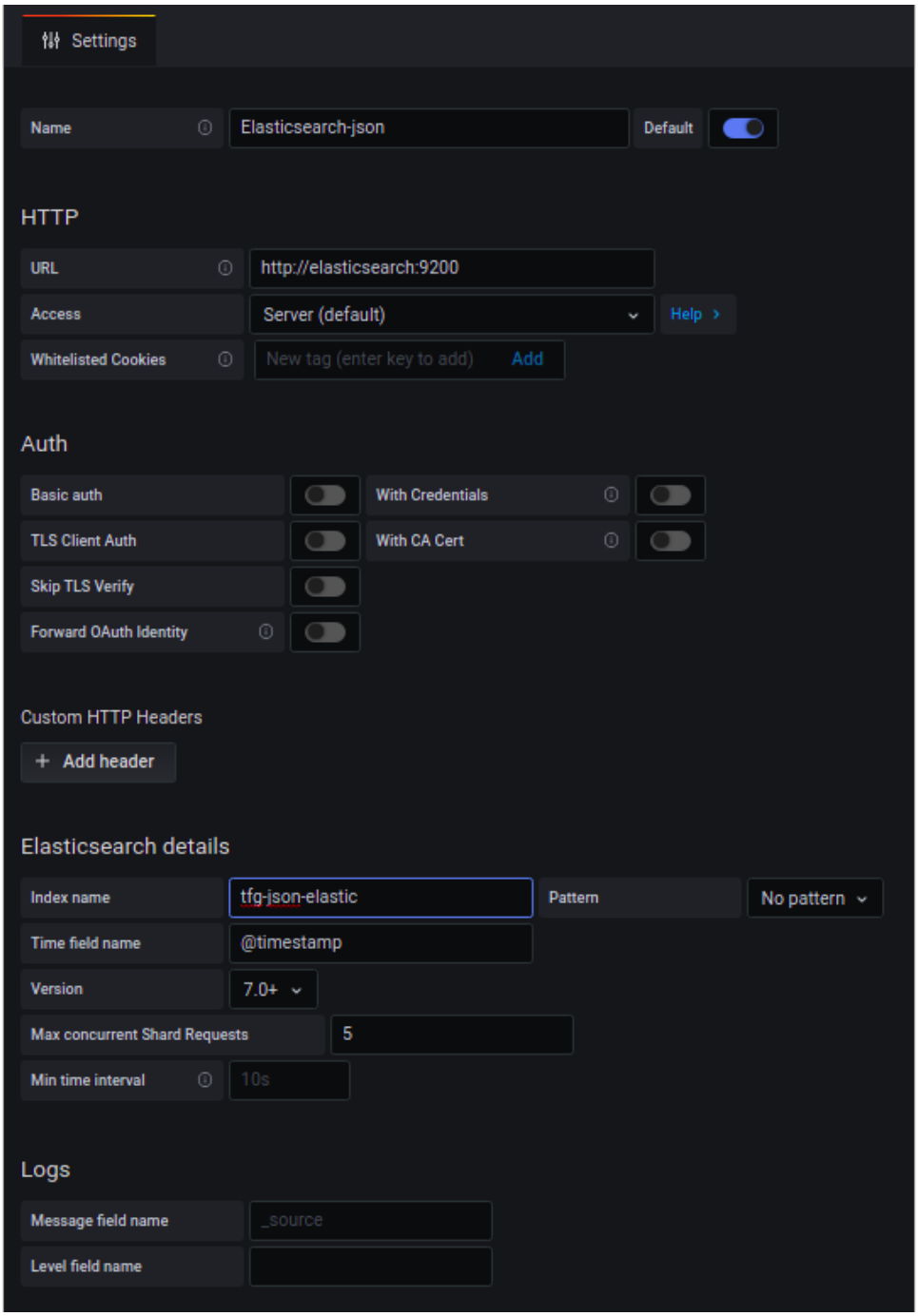


Figura 3.4. (a) Creación de Datasource con motor Elasticsearch e índice tfg-json-elastic, para crear un Dashboard de los datos subidos a Elasticsearch mediante Python con un formato de archivo JSON.

3.4.5 Análisis de datos CSV

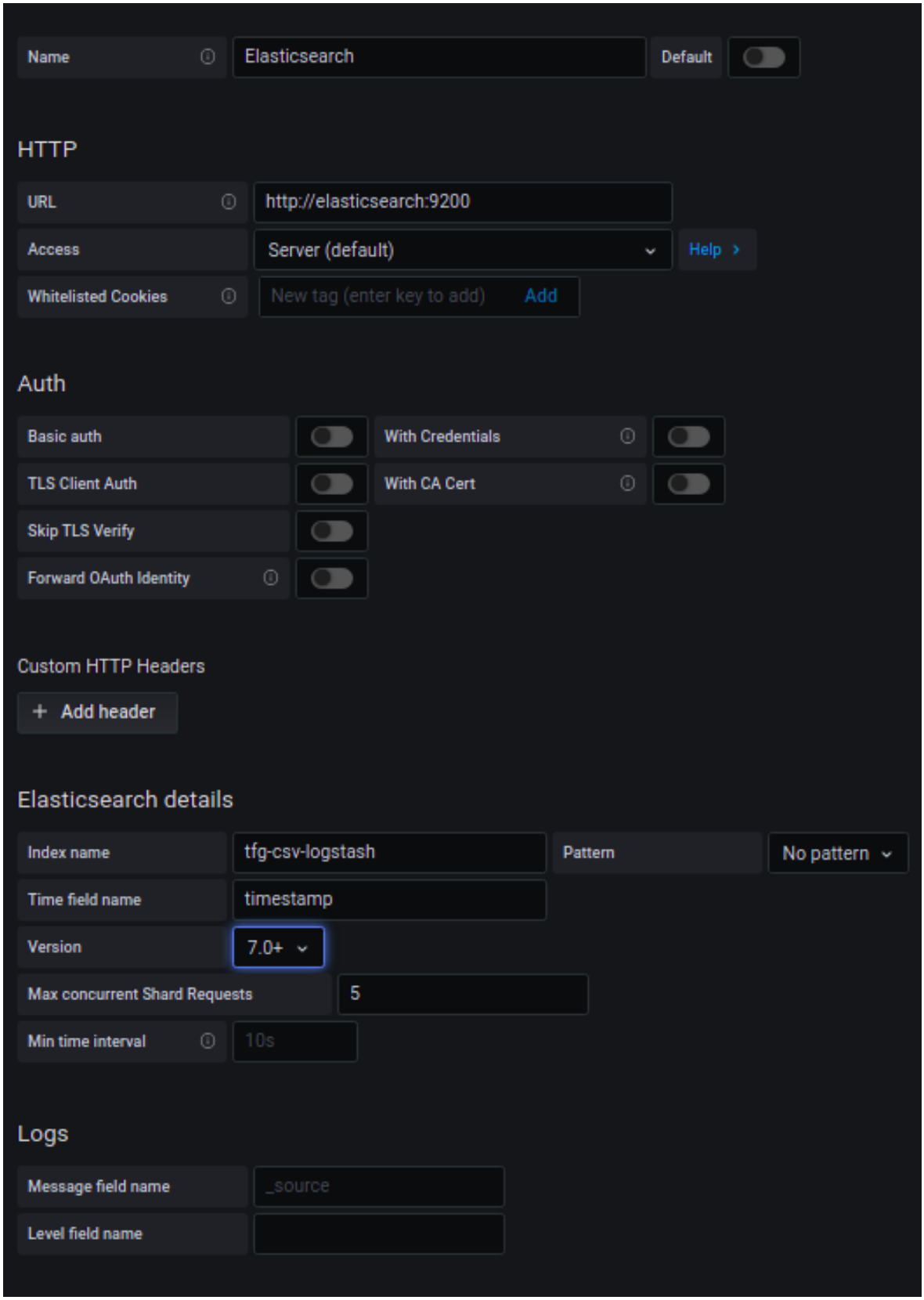


Figura 3.4. (b) Creación de Datasource con motor Elasticsearch e índice tfg-csv-logstash, para crear un Dashboar de los datos subidos a Elasticsearch mediante Logstash, con un archivo CSV.

4 Integración, pruebas y resultados

Como se ha explicado en capítulos anteriores, la integración de este proyecto es realizada en su mayoría con Docker-Compose. Esta herramienta nos ha permitido establecer y concertar entre sí los servicios de Elasticsearch, Grafana y Dejavu, de manera muy fácil y rápida. Por otro lado, con Logstash, se ha decidido instalarlo aparte en el equipo. Los programas de Python de conversión de archivos se han realizado desde el entorno PyCharm.

Ahora se pasa a explicar el funcionamiento de los servicios webs y de cómo los hemos integrado. Empezando por el Docker-Compose.yml, Anexo C.A. En este archivo se puede encontrar la versión en la que está, los servicios que se van a instalar y sus configuraciones. Debajo de cada servicio se encuentran las distintas opciones de su configuración, como URL, puertos o conexiones a otros programas.

Una vez que está el entorno preparado, se necesita verificar su correcto funcionamiento y ejecutar los programas en su orden correspondiente. Esto vamos a realizarlo a continuación con las pruebas.

4.1 Pruebas

Como hemos mencionado antes, los archivos que se usan para las pruebas son archivos DUMP muy reducidos creados a partir de los archivos originales, para verlos ir a los anexos A. Se han creado dos archivos diferentes para que el procesamiento de los datos y las pruebas sean lo más exactas, verídicas y similares al trabajo superior que se acabará realizando.

Una vez creado el Docker-compose.yml (Anexo C.A), instalado Logstash y creado el logstash.conf (Anexo C.B), se necesita ejecutar el Docker-compose.yml y acceder a cada servicio web para comprobar que funcionan correctamente. En el propio Docker-compose.yml se puede observar que las respectivas urls son: http:// IP local +: el respectivo puerto

- IP local = http://localhost

- Puertos:
  - Elasticsearch = :9200
  - Dejavu = :1538
  - Grafana= :3000

Para ejecutar Docker Compose se usa el siguiente comando:

$ docker-compose up -d

Este comando hay que ejecutarlo en el mismo directorio donde esté guardado el archivo Docker-compose.yml

4.1.1 Transformación de Archivos DUMP

Ahora se va a ejecutar los programas de transformación de archivos uno por uno, obteniendo los primeros resultados.

DUMP a JSON (Anexo B.A.)

$ python3 DumpToJsonForElastic.py

Medición de tiempo de conversión de archivos en

Tiempo de ejecución archivos de prueba: 0.0012049674987792969 segundos

Tiempo de ejecución archivos de datos: 62,6468920 segundos = 1.044 minutos

DUMO a CSV Anexo B.B**.**

$ python3 DumpToCsvForLogstash.py

Medición de tiempo de conversión de archivos

Tiempo de ejecución archivos de prueba: 0.00060272216796875 segundos

Tiempo de ejecución archivos de datos: 22.88214111 segundos = 0.3813666 minutos

Se puede comprobar que son correctos, y que los tres archivos disponen de las mismas ocho líneas de datos.

4.1.2 Subida del contenido de los archivos a Elasticsearch

Ahora se sube el contenido a Elasticsearch:

Archivo JSON

Para subir el archivo JSON necesitamos ejecutar el programa: UploadToElasticsearch.py Anexo C.C.

$python3 UploadToElasticsearch.py

- Tiempo de ejecución archivos de prueba: 0.45963239669799805 segundos.
- Tiempo de ejecución archivos de datos: 4.7595 Minutos, (Con 8 GB de RAM no se puede terminar la ejecución del programa).

Una vez haya terminado, comprobamos que se haya creado el índice con el comando:

$ curl -XGET localhost:9200/_cat/indices?v

Si no ha habido problemas, dentro de Dejavu, introduciendo el índice que, creado con este programa, se comprueba que los datos son correctos.

Tiempo Total Transformar y subir datos (en segundos):

- Total, subir archivos de prueba a Elasticsearch mediante Python: 0.460 + 0,4596 = 0,91 segundos.
- Total, subir archivos a Elasticsearch mediante Python: 1.044 + 4.7595 = 5,8035 minutos.

Archivo CSV

Para subir el archivo CSV vamos a utilizar el logstash.conf como hemos explicado anteriormente:

Primero necesitamos comprobar que Logstash y Elasticsearch están operativos con los comandos de control de web Services. Cuando comprobemos que están operativos, vamos a la carpeta desde donde ejecutar Logstash, y utilizamos el comando:

$ sudo ./logstash -f logstash.conf

Ahora se espera a que se terminen de subir los datos, mientras se puede observar su procesamiento por terminal. De igual modo que antes, se comprueba que la creación del índice desde terminal, entramos en Dejavu para ver la conexión con este nuevo índice para la verificación de que el proceso haya sido correcto.

- Tiempo de ejecución archivos de prueba: 18.08 segundos.
- Tiempo de ejecución archivos de datos: 11,998 minutos.

Este tiempo lo hemos medido mediante el comando time de Linux:

$ sudo time ./logstash -f logstash.conf

Tiempo Total Transformar y subir datos:

- Total, subir archivos de prueba a Elasticsearch mediante Logstash: 18.08 + 0.0006 = 18,08 segundos
- Total, subir archivos a Elasticsearch mediante Logstash: 0.38136 + 11,998 = 12,3 minutos

Tiempo Total Transformar y subir datos:

- Total, subir archivos de prueba a Elasticsearch mediante comandos de Elasticsearch: 5,8 minutos
- Total, subir archivos a Elasticsearch mediante comandos de Elasticsearch: 12,3 minutos

Grafana

Desde Grafana se han de crear los distintos Datasource de Elasticsearch, uno para cada índice. Esto se realiza desde la dirección: http://grafana:3000

Una vez creados, generamos un Dashboard nuevo para cada Datasource, en el que se nos muestran los resultados, según los campos seleccionados. Se observan los resultados en las Figuras 4.1 y 4.2:

Figura 4.1 Muestra de resultados del fichero de prueba para el índice tfg-json-elastic.

Figura 4.2 Muestra de resultados del fichero de prueba para el índice tfg-csv-logstash.

Gracias a estas dos imágenes, se comprueba que se han subido el mismo número de datos de las dos maneras para ambos ficheros (como son ficheros de datos de prueba, sabemos que el total tiene que ser 8, como se indica). Anteriormente se había comprobado directamente desde Dejavu de manera más específica, pero siempre viene bien hacer múltiples comprobaciones. Esta veracidad es también apreciable en las Figuras 4.5.(a) y 4.5.(b) donde se manipulan los datos de una manera algo más específica.

En las siguientes (figuras 4.3 y 4.4) podemos observar una sencilla modificación de los parámetros para mostrar otro tipo de datos y de otra manera. El resultado para los distintos índices aparece en las Figuras 4.5.(a). y 4.5.(b).

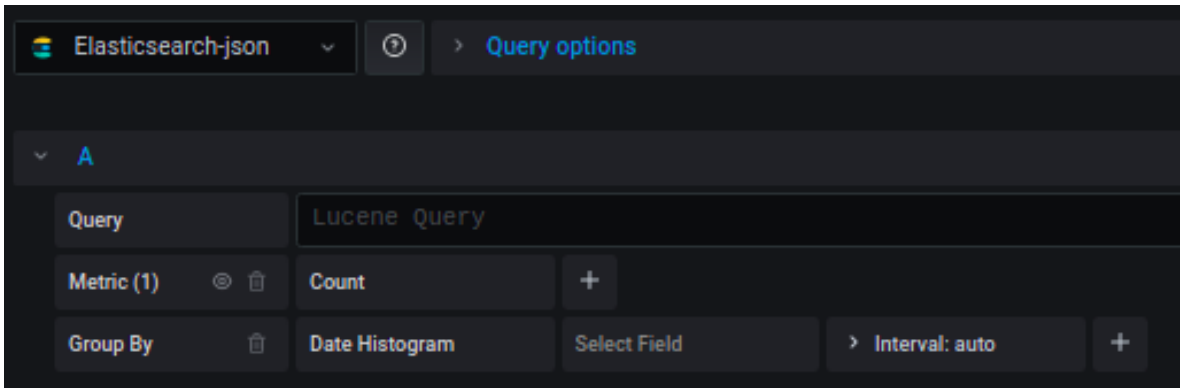
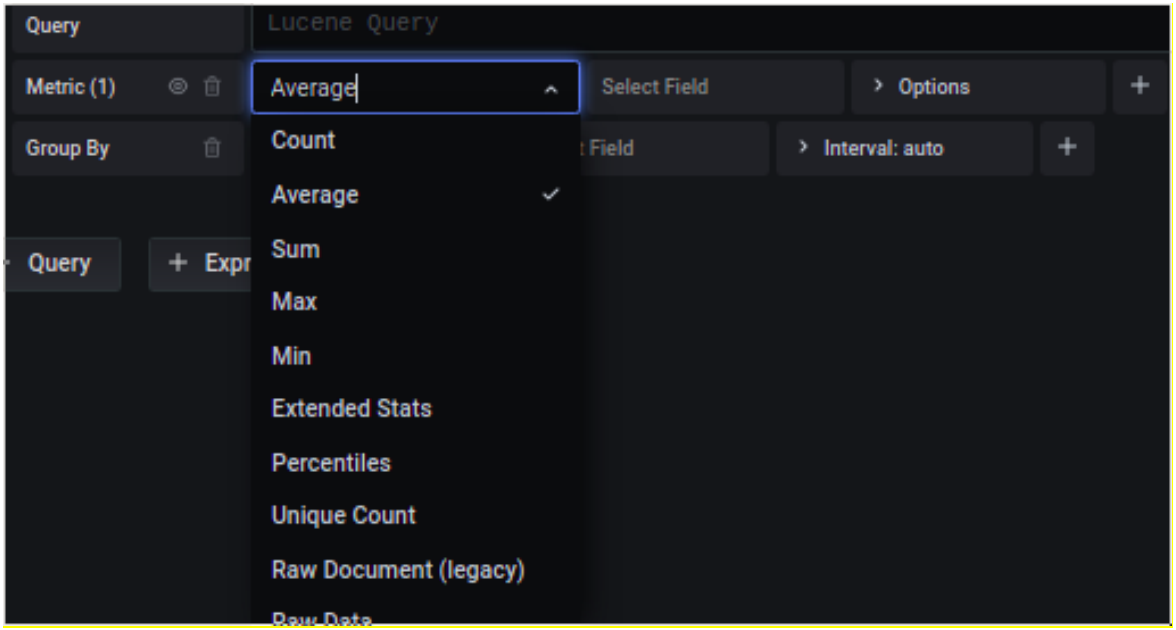


Figura 4.3. Foto donde se observan las opciones de los parametros, en este es el caso predeterminado



. Figura 4.4. Muestra las distintas operaciones seleccionables para las distintas columnas de los datos.

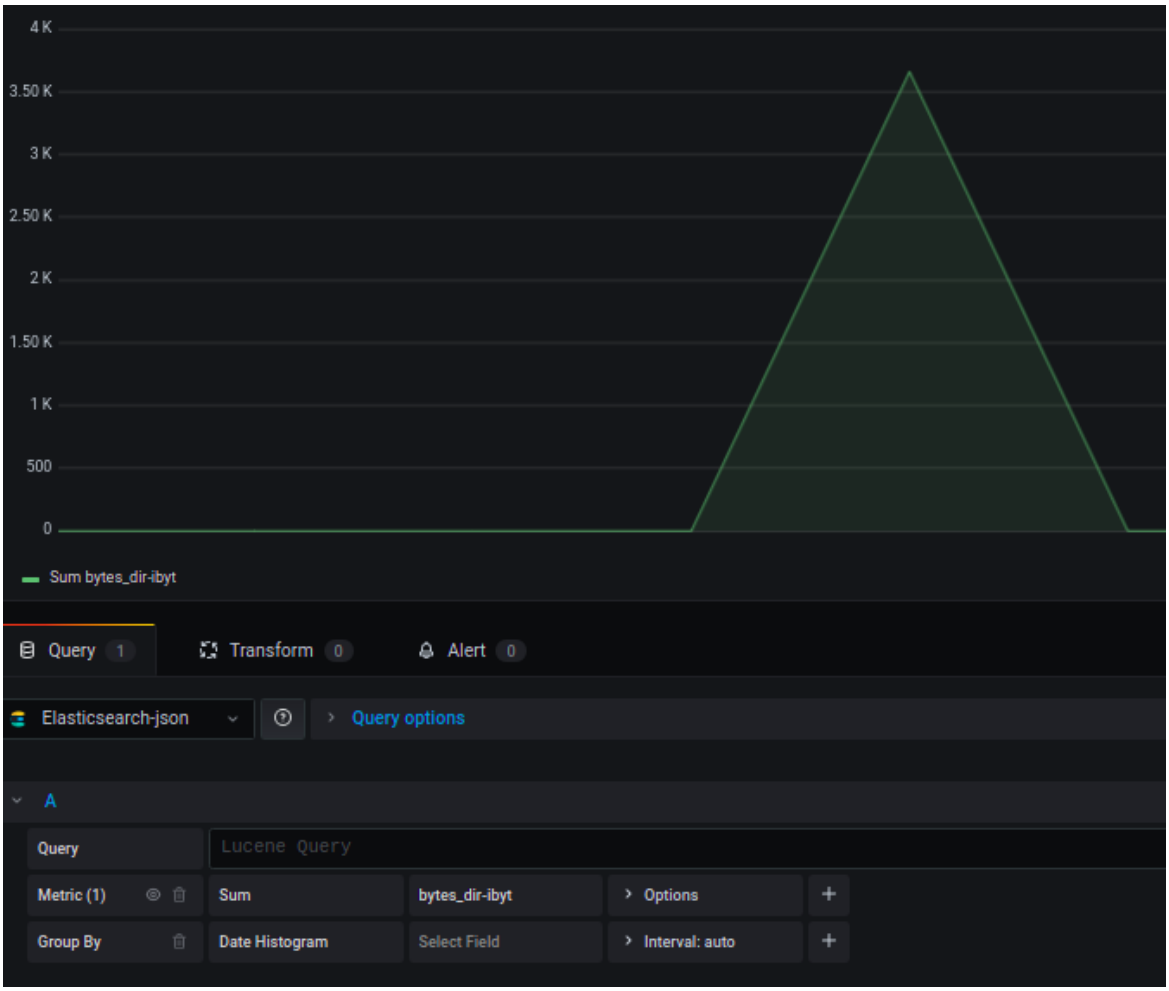


Figura 4.5. (a) Se puede observar el resultado de las opciones y operaciones enseñadas, con la operación suma, que realiza la suma de todas las columnas, del campo bytes_dir-ibyt.

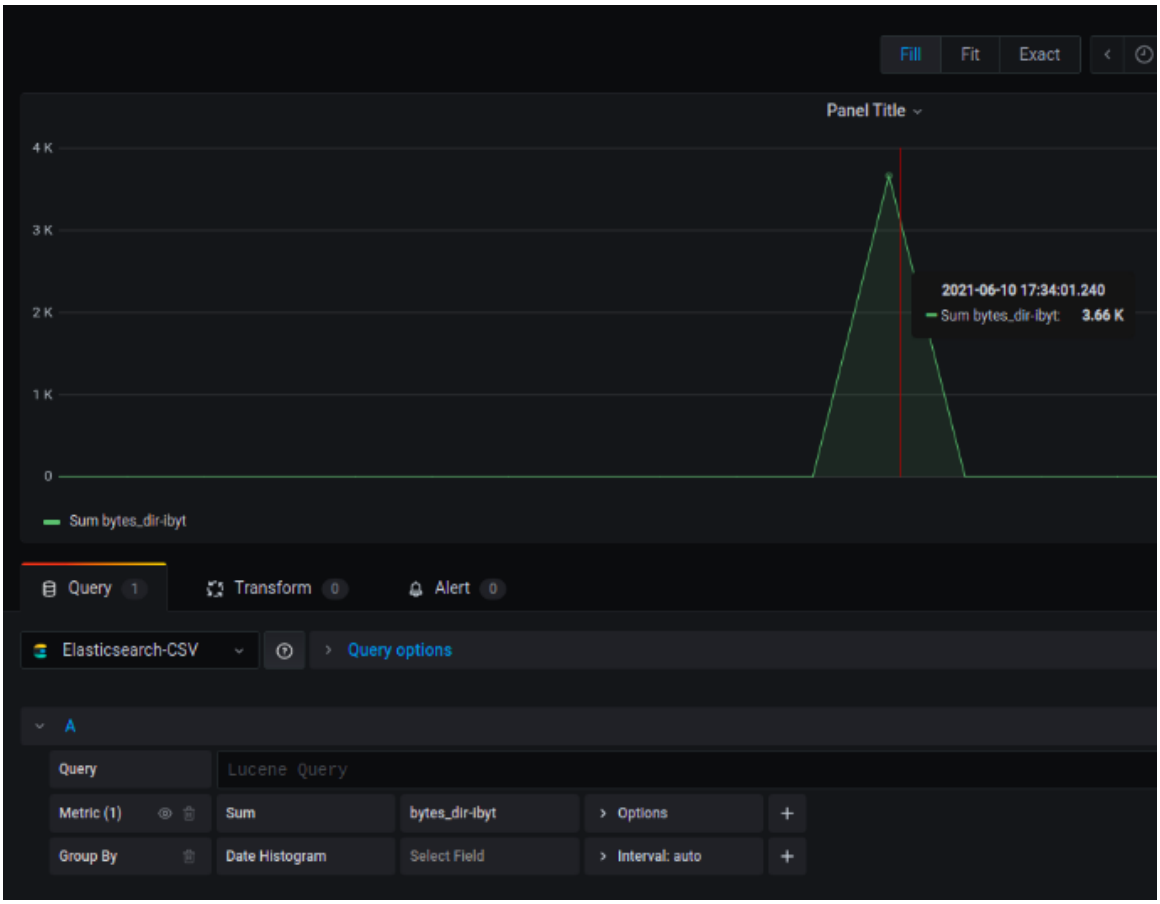


Figura 4.5 (b) Comparando las dos fotos (los dos índices) se observa que son iguales, obteniendo 3.66K como resultado total de la suma en el campo bytes_dir-ibut en ambos, reforzando la veracidad de los procesos.

4.1.3 EXPERIMENTOS

Se ha experimentado subiendo a Elasticsearch con Logstash un fichero JSON, este experimento ha resultado sencillo de hacer, con los tiempos esperados. Vería relativamente poco comparado con un archivo CSV. El proceso es también idéntico, resulta sencillo realizar el Logstash.conf. Como la transformación de datos a CSV es más veloz y es un archivo más sencillo de construir, así como más versátil para mover los datos, se considera que es mejor esta manera de subir los archivos con Logstash antes que con JSON.

Se han experimentado distintas maneras de medir el tiempo, por comando en la terminal, en el propio programa de Pyhton, y con cronómetro. Se ha observado que la forma más precisa para medir el tiempo es en el propio programa. Cuando no ha sido posible, como en el caso del fichero Logstash.conf, se ha usado el comando time en la terminal. El formato de cronometrar se ha descartado completamente.

4.2 RESULTADOS.

Siguiendo los pasos anteriores con los ficheros completos obtenemos los siguientes resultados de los Datasource de Grafana.

Vamos a analizar principalmente, la calidad de los datos, los errores generados, la complejidad del proceso, las ventajas y desventajas de este, y el tiempo que tarda el proceso completo.

JSON:
Proceso: El proceso de subir un JSON a Elasticsearch es más complejo. Una vez realizado esto, lo siguiente es similar para el resto metodologías. Tiene la ventaja de, necesitar menos programas que el otro proceso.

- Ventajas: Las ventajas de este método son que es mucho más rápido.

- Desventajas: Las desventajas son que utiliza muchísima RAM para poder subir el archivo a Elasticsearch tan rápidamente, así la dificultad de programar la subida del fichero.

- Complejidad: Este proceso requiere de conocimientos de Python amplios y moderadamente avanzados.

- Errores o dificultad: Los principales problemas han sido debido a la excesiva RAM que requiere para un gran fichero de datos como es el que se nos proporciona, para las pruebas ha funcionado siempre correctamente. La principal dificultad se encuentra en el método bulk de Python para Elasticsearch.

- Tiempo de generación del proceso completo: 5,8035 minutos.

CSV:
Proceso: Este proceso resulto sencillo de aprender y utilizarse. Además, Logstash es muy visual lo que facilita el trabajo para el programador, es un programa especializado para subir datos a Elasticsearch.

- -ventajas: La ventaja de este proceso reside en la visualización de la subida de datos, así como la facilidad de subir datos con Logstash.

- -desventajas: Las desventajas es que el proceso de subida de datos es mucho más lento que el mencionado anteriormente.

- -complejidad: Es más sencillo y se encuentra documentación más clara, así como más abundante, de cómo usar y subir datos con Logstash, lo que hace más sencillo aprender su funcionamiento y utilizarlo acorde a nuestros intereses.

- -Errores o dificultad: Los errores principalmente se cometieron en el archivo logstash.conf, por temas de direcciones y donde tiene que estar este archivo colocado. Los demás errores pueden considerarse pura rutina de programador, como necesitar parsear variables u olvidarse una coma.

- Tiempo de generación del proceso completo: 12,3 minutos

5 Conclusiones y trabajo futuro

Este capítulo trata las conclusiones para resumir el trabajo completo. Recordando al lector que se ha realizado y más por encima como, para conseguir una lectura provechosa. Además de los dos procesos estudiados, este capítulo también informará a cerca de otros trabajos de investigación posibles de realizar desde este punto del TFG.

5.1 Conclusiones

Con toda esta información, y una vez comprobada la veracidad de los datos en el Dashboard (los resultados de las consultas que muestran los gráficos deben ser iguales), y con esto de los procesos, se llega a una conclusión bastante acertada, ningún proceso es mejor que el otro al cien por cien, cada uno funciona mejor según las capacidades y recursos disponibles. Las conclusiones y resultados de tiempos obtenidos en este TFG son únicamente replicables o muy similares con un ordenador similar al que se ha utilizado para ejecutar los programas y procesos, el tiempo va ligado a la potencia de este, a mayor potencia, menos tardarán los procesos en completarse.

Si se dispone de un ordenador muy potente, o, en consecuencia, con un ordenador potente, el tiempo del que se dispone es corto, optaremos por la primera opción, donde se utiliza una inmensidad de memoria RAM para subir los datos a Elasticsearch con Python. Debido a limitaciones software se generaron grandes problemas, en concreto con la memoria RAM, el proceso realizaba un kill porque una memoria de 8 gigas de RAM colapsa. Fue necesario buscar otro equipo más potente donde poder instalar el entorno para ejecutar este proceso y el resto para repetir las mediciones. El proceso es muy rápido, el doble que el obtenido subiendo los archivos por Logstash, pero a cambio, este otro proceso es muy amigable con los recursos del equipo, y muy óptimo.

Por otro lado, el proceso de transformación de datos es casi el triple de grande para convertir a JSON. Esto tiene una menor relevancia, porque son tiempos bastante más pequeños, y se sigue usando una enorme cantidad de datos de golpe. Aunque tenga menor relevancia, se comentara en trabajo futuro.

El proceso de subir los datos a Elasticsearch por Python, a diferencia de Logstash, se contempla sin control de errores, ni se muestran los datos por la terminal. Dependiendo del control de errores y de datos realizado, esto podría alargar bastante el tiempo de este proceso. Es muy posible que imprimiendo cada dato que se va subiendo a Elasticsearch por terminal, como hace Logstash, los dos tiempos se igualen bastante.

También se comprueba que Logstash es un método muchísimo más limpio, y eficiente a la hora de gestionar recursos. Tarda más, pero es mucho más organizado y visual para el programador, así como sencillo de realizar. Esto siempre es un punto a favor, el orden y la limpieza en los trabajos profesionales son fundamentales, hasta determinantes. Además, la curva de aprendizaje de Logstash, resulto ser más sencilla de lo esperado, haciendo del trabajo y aprendizaje agradable.

Gracias a Docker, se ha agilizado el proceso enormemente, esta herramienta es algo fundamental hoy en día y debería de utilizarse mucho más. Esta herramienta se retroalimenta con el número de personas que la usan, es decir, cuanta más gente la utilice más potente será, por lo que ha sido un completo acierto trabajar con ella para realizar este TFG.

La Tabla 5.1 resume completamente los dos procesos y sus mejores y peores cualidades.

	Tiempo	Recursos	Dificultad	Otros
Python	Rápido	Excesivo	Complejo	-
Logstash	Lento	Optimo	Sencillo	Es muy visual, dando sensación de profesionalidad

Tabla 5.1. Tabla Resumen de los Procesos, donde podemos compararlos rápidamente y apreciar los puntos fuertes y flojos de cada uno.

Gracias esta tabla, la conclusión de este TFG es que es mucho mejor usar Logstash. A no ser que se disponga de un tiempo excesivamente pequeño, o de un fichero absolutamente descomunal.

5.2 Trabajo futuro

Este TFG, es algo muy útil, y con otros programas competidores para Grafana como podría ser Power BI, y para Elasticsearch, como podrían ser otra base de datos. Sería algo muy interesante comparar los tiempos y el uso de recursos de Elasticsearch y Grafana, con su mayor competencia existente. Esto podría enseñarnos cuando es mejor utilizar cada tecnología, obteniendo los mejores resultados en la mayoría de los datos.

Se podría hacer con cualquier otra tecnología que sea competente o una potencial competidora para Grafana y Elasticsearch. De este modo, acumulando la información respectiva y haciendo un análisis intensivo de los procesos más específicos para cada programa, así como los que tengan en común, incluyendo el estudio de distintos archivos de datos. Se puede realizar un programa donde un usuario introduzca sus intereses, y el programa responda que método es el más adecuado, o de varias opciones igual de óptimas.

Referencias

[1] D. A. Perez Aguilar, "¿La información es poder?", Blogs.deusto.es, 2021. [Online]. Available: https://blogs.deusto.es/master-informatica/la-informacion-es-poder/.

[2] P. E. Ceruzzi. Historia de la informática. Open Mind. Recuperado de https://www.bbvaopenmind.com/wp-content/uploads/static/pdf/109-127_PAUL_E._CERUZZI_ESP_R.pdf. 2018.

[3] A Silberschatz, H.F Korth, S. Sudarshan. Fundamentos de bases de datos. McGraw Hill. 2002.

[4] “Historia de las Bases de Datos", Historia de la Informática, 2011.

[5] M. Guerrero, "La Quinta Revolución Industrial", Kaizen, 2018.

[6] "JSON", Json.org, 2021. [Online]. Available: https://www.json.org/json-en.html.

[7] J. Ansa, “Archivos de volcado de memoria (Dump Files)", Geeks.ms, 2008.

[8] Y. Shafranovich, "CSV", Datatracker.ietf.org, 2005. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4180#page-2.

[9]Configuring Logstash | Logstash Reference [7.13] | Elastic, Elastic.co, 2021. [Online]. Available: https://www.elastic.co/guide/en/logstash/current/configuration.html.

[10] "Grafana Screencasts Episode 9", Youtube.com, 2021. Available: https://www.youtube.com/watch?v=d6KicssNzxM.

[11] B. Berman, “Logz.io, 2021”. Available: https://logz.io/blog/monitoring-dockerized-elk-stack.

[12] R. Ibarra, "Usar Logstash, Elasticsearch y Grafana para monitorear servicios", Linkedin.com, 2021. [Online]. Available: https://www.linkedin.com/pulse/usar-logstash-elasticsearch-y-grafana-para-monitorear-ricardo-ibarra/?trk=public_profile_article_view.

Glosario

RDBSM Bases de Datos Relacionales

OODBMS Bases de Datos Orientadas a Objetos

IBM International Business Machines Corporation

API Application Programming Interface

DUMP Volcado de Memoria

CSV Comma-Separated Values

JSON JavaScript Object Notation

JSONL JSON Lines

ELK Elasticsearch, Logstash & Kibana

RAM Random-Access Memory

Anexos

A Archivos de Datos

A.A TestA.dump

```
2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-
08T16:15:00-
05:00,6,192.168.29.3,201.116.65.2,201.116.168.245,30758,443,0,0,
9,0,2855,0,1,7633,3,317
2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-
08T16:15:00-
05:00,17,192.168.29.3,62.2.21.168,201.116.168.140,12193,53,0,0,1
,0,83,0,1,0,0,83
2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-
08T16:15:00-
05:00,17,192.168.29.3,201.116.168.140,205.251.197.157,18367,53,0
,0,1,0,71,0,1,0,0,71
2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-
08T16:15:00-
05:00,17,172.22.201.34,172.27.15.191,172.22.201.80,52108,53,219,
77,1,0,85,0,1,0,0,85
```

A.B TestB.dump

```
2020-09-08T16:20:00-05:00,2020-09-08T16:20:00-05:00,2020-09-
08T16:20:00-
05:00,17,192.168.29.3,201.116.168.140,172.253.214.105,53,41822,0
,0,1,0,177,0,1,0,0,177
2020-09-08T16:20:00-05:00,2020-09-08T16:20:00-05:00,2020-09-
08T16:20:00-
05:00,6,172.16.201.213,172.16.216.123,172.16.215.8,8081,58446,86
8,46,3,0,164,0,3,0,0,54
2020-09-08T16:20:00-05:00,2020-09-08T16:20:00-05:00,2020-09-
08T16:20:00-
05:00,17,192.168.29.3,172.17.201.81,172.17.106.229,53,56879,0,0,
1,0,158,0,1,0,0,158
2020-09-08T16:20:00-05:00,2020-09-08T16:20:00-05:00,2020-09-
08T16:20:00-
05:00,17,172.22.201.34,172.22.2.6,172.22.201.80,53363,53,219,77,
1,0,64,0,1,0,0,64
```

B Conversión de Archivos

B.A Conversión a JSON

```
# Transform .dump DATA into .JSON
"""
@author: Hoyo Bravo, Guillermo
"""
import json
import time

LOG_FILENAME = 'LoggingDebug'

def main():
    inicio = (time.time())
    #Contador
    counter = 0
    counter2 = 0
    #Listo of dictionaries
    LDictionary = []
    #Files of data dictionarys
    #f = open('nfcapd.202009081610.dump', 'r')
    #fi = open('nfcapd.202009081615.dump', 'r')
    f = open('testA.dump', 'r')
    fi = open('testB.dump', 'r')
    #Spliting first file in 1 line for 1 dictioanry
    for linea in f:
        campos = linea.split(',')
        '''
        DEBUG LISTA
        for element in campos:
            print(element)
        '''
        '''
        DEBUG LEN LISTA
        print(len(campos))
        '''
        #CHECK
        if len(campos) != 19:
            print('ERROR splitting file information')
            return -1
        #Insert Line into dictionary
        #Dictionary Structure
        Dictionary = {
            'timestamp-ts1': campos[0],
            'timestamp-ts2': campos[1],
            'timestamp-ts3': campos[2],
            'pr': int(campos[3]),
            'ips-ra': campos[4],
            'ips-sa': campos[5],
            'ips-da': campos[6],
            'ports-sp': campos[7],
            'ports-dp': campos[8],
            'package_action-iin': int(campos[9]),
            'package_action-out': int(campos[10]),
            'package_dir-ipkt': int(campos[11]),
            'package_dir-opkt': int(campos[12]),
            'bytes_dir-ibyt': int(campos[13]),
            'bytes_dir-obyt': int(campos[14]),
```

```
            'fl': int(campos[15]),
            'rates-bps': int(campos[16]),
            'rates-pps': int(campos[17]),
            'rates-bpp': int(campos[18])
        }
        '''
        DEBUG ADDED INFO TO DICTIONARY
        print(Dictionary.items())
        '''
        #Adding each Dictionary to the List
        LDictionary.append(Dictionary)
        counter = counter + 1

    #CHECH Dictionary Length?
    if len(LDictionary) != counter:
        print('ERROR of the list Length')
        return -2

    '''
    DEBUG ELEMENTS OF THE LIST
    for element in LDictionary:
        print(element)
    '''

    #Open Second File to Transform
    for linea in fi:
        componentes = linea.split(',')
        '''
        DEBUG LISTA
        for element in componentes:
            print(element)
        '''
        '''
        DEBUG LEN LISTA
        print(len(componentes))
        '''
        #CHECK
        if len(componentes) != 19:
            print('ERROR splitting file information')
            return -3

        #Dictionary Structure
        Dictionary2 = {
            'timestamp-ts1': componentes[0],
            'timestamp-ts2': componentes[1],
            'timestamp-ts3': componentes[2],
            'pr': int(componentes[3]),
            'ips-ra': componentes[4],
            'ips-sa': componentes[5],
            'ips-da': componentes[6],
            'ports-sp': componentes[7],
            'ports-dp': componentes[8],
            'package_action-iin': int(componentes[9]),
            'package_action-out': int(componentes[10]),
            'package_dir-ipkt': int(componentes[11]),
            'package_dir-opkt': int(componentes[12]),
            'bytes_dir-ibyt': int(componentes[13]),
            'bytes_dir-obyt': int(componentes[14]),
```

```
            'fl': int(componentes[15]),
            'rates-bps': int(componentes[16]),
            'rates-pps': int(componentes[17]),
            'rates-bpp': int(componentes[18])
        }
        '''
        DEBUG ADDED INFO TO DICTIONARY
        print(Dictionary2.items())
        '''
        #Adding each Dictionary to the List
        LDictionary.append(Dictionary2)
        counter2 = counter2 + 1


    #CHECH Dictionary Length?
    if len(LDictionary) != counter2 + counter:
        print('ERROR of the list Length')
        return -4

    '''
    DEBUG ELEMENTS OF THE LIST
    for element in LDictionary:
        print(element)
    '''
    #CHECK LAST ELEMENT OF THE FILE
    #print(LDictionary[counter+counter2-1])

    #Pasar Lista a archivo.JSON
    with open('TestElasticsearch.json', 'w') as fil:
        for element in LDictionary:
            json.dump(element, fil)
            fil.write("\n")

            '''
            DEBUG ADDED LINE TO JSON
            printf(Added line: element)
            '''

        print('The files have been Exportted')

    f.close()
    fi.close()
    fin = (time.time())
    print("Tiempo Json para ElasticSearch")
    print(fin-inicio)
    return

if __name__ == '__main__':
    main()
```

B.B Conversión a CSV

```
# Transform .dump DATA into .JSON
"""
@author: Hoyo Bravo, Guillermo
"""
import time
LOG_FILENAME = 'LoggingDebug'
def main():
    inicio = (time.time())
    #Contador
    counter = 0
    counter2 = 0
    #Listo of dictionaries
    LDictionary = []
    #Files of data dictionarys
    #f = open ('nfcapd.202009081610.dump', 'r')
    #fi = open ('nfcapd.202009081615.dump', 'r')
    f = open('testA.dump', 'r')
    fi = open('testB.dump', 'r')
    #Spliting first file in 1 line for 1 dictioanry
    for linea in f:
        campos = linea.split(',')
        '''
        DEBUG LISTA
        for element in campos:
            print(element)
        '''
        '''
        DEBUG LEN LISTA
        print(len(campos))
        '''
        #CHECK
        if len(campos) != 19:
            print('ERROR splitting file information')
            return -1
        #Insert Line into dictionary
        #Dictionary Structure
        Dictionary = [
            campos[0],
            campos[1],
            campos[2],
            int(campos[3]),
            campos[4],
            campos[5],
            campos[6],
            campos[7],
            campos[8],
            int(campos[9]),
            int(campos[10]),
            int(campos[11]),
            int(campos[12]),
            int(campos[13]),
            int(campos[14]),
            int(campos[15]),
            int(campos[16]),
            int(campos[17]),
            int(campos[18])
        ]
```

```
    '''
    DEBUG ADDED INFO TO DICTIONARY
    print(Dictionary.items())
    '''
    #Adding each Dictionary to the List
    LDictionary.append(Dictionary)
    counter = counter + 1
#CHECH Dictionary Length?
if len(LDictionary) != counter:
    print('ERROR of the list Length')
    return -2
'''
DEBUG ELEMENTS OF THE LIST
for element in LDictionary:
    print(element)
'''
#Open Second File to Transform
for linea in fi:
    componentes = linea.split(',')
    '''
    DEBUG LISTA
    for element in componentes:
        print(element)
    '''
    '''
    DEBUG LEN LISTA
    print(len(componentes))
    '''
    #CHECK
    if len(componentes) != 19:
        print('ERROR splitting file information')
        return -3
    #Dictionary Structure
    Dictionary2 = [
        componentes[0],
        componentes[1],
        componentes[2],
        int(componentes[3]),
        componentes[4],
        componentes[5],
        componentes[6],
        componentes[7],
        componentes[8],
        int(componentes[9]),
        int(componentes[10]),
        int(componentes[11]),
        int(componentes[12]),
        int(componentes[13]),
        int(componentes[14]),
        int(componentes[15]),
        int(componentes[16]),
        int(componentes[17]),
        int(componentes[18])
    ]
    '''
    DEBUG ADDED INFO TO DICTIONARY
    print(Dictionary2.items())
    '''
```

```
        #Adding each Dictionary to the List
        LDictionary.append(Dictionary2)
        counter2 = counter2 + 1
    #CHECH Dictionary Length?
    if len(LDictionary) != counter2 + counter:
        print('ERROR of the list Length')
        return -4
    '''
    DEBUG ELEMENTS OF THE LIST
    for element in LDictionary:
        print(element)
    '''
    #CHECK LAST ELEMENT OF THE FILE
    #print(LDictionary[counter+counter2-1])
    #Pasar Lista a archivo.JSON
    with open('TestLogstashCSV.csv', 'w') as fil:
        fil.write("timestamp-ts1,timestamp-ts2,timestamp-
ts3,pr,ips-ra,ips-sa,ips-da,ports-sp,ports-dp,package_action-
iin,package_action-out,package_dir-ipkt,package_dir-
opkt,bytes_dir-ibyt,bytes_dir-obyt,fl,rates-bps,rates-pps,rates-
bpp\n")
        for element in LDictionary:
            n = 0
            for x in element:
                fil.write(str(x))
                if n < 18:
                    fil.write(",")
                    n = n+1
                    #print(n)
            fil.write("\n")
            '''
            DEBUG ADDED LINE TO JSON
            printf(Added line: element)
            '''
        print('The files have been Exportted')
    f.close()
    fi.close()
    fin = (time.time())
    print("Tiempo de Dump a CSV")
    print(fin-inicio)
    return
if __name__ == '__main__':
    main()
```

C Archivos de Configuración

C.A Archivo Docker-Compose.yml

```
version: '3'

services:
  grafana:
    image: grafana/grafana
    ports:
      - 3000:3000
    links:
      - Elasticsearch

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch-oss:7.0.1
    container_name: elasticsearch
    environment:
      - discovery.type=single-node
      - http.port=9200
      - http.cors.enabled=true
      - http.cors.allow-
origin=http://localhost:1358,http://127.0.0.1:1358
      - http.cors.allow-headers=X-Requested-With,X-Auth-
Token,Content-Type,Content-Length,Authorization
      - http.cors.allow-credentials=true
      - bootstrap.memory_lock=true
      - 'ES_JAVA_OPTS=-Xms512m -Xmx512m'
    ports:
      - '9200:9200'
      - '9300:9300'

  dejavu:
    image: appbaseio/dejavu:3
    container_name: dejavu
    ports:
      - '1358:1358'
    links:
      - Elasticsearch
```

C.B Archivo Logstash.conf

```
input{
    file{
        path =>
"/home/lilg8b/Downloads/TFGnew/TFG/FicherosDump/TestLogstashCSV.csv
"
        start_position => "beginning"
        sincedb_path => "/dev/null"
    }
}
filter{
    csv{
        separator => ","
        columns => ["timestamp-ts1","timestamp-ts2","timestamp-
ts3","pr","ips-ra","ips-sa","ips-da","ports-sp","ports-
dp","package_action-iin","package_action-out","package_dir-
ipkt","package_dir-opkt","bytes_dir-ibyt","bytes_dir-
obyt","fl","rates-bps","rates-pps","rates-bpp" ]
    }
    mutate {convert => ["pr", "integer"]}
    mutate {convert => ["package_action-iin", "integer"]}
    mutate {convert => ["package_action-out", "integer"]}
    mutate {convert => ["package_dir-ipkt", "integer"]}
    mutate {convert => ["package_dir-opkt", "integer"]}
    mutate {convert => ["bytes_dir-ibyt", "integer"]}
    mutate {convert => ["bytes_dir-obyt", "integer"]}
    mutate {convert => ["fl", "integer"]}
    mutate {convert => ["rates-bps", "integer"]}
    mutate {convert => ["rates-pps", "integer"]}
    mutate {convert => ["rates-bpp", "integer"]}
}
output{
    elasticsearch{
        hosts => ["localhost:9200"]
        index => "tfg-csv-logstash"
        document_type => "cvs_tfg_data"
    }
    stdout{}
}
```

C.C Subir JSON a Elasticsearch con Python

```
#Importamos la librería JSON
import json
#Importamos Elasticsearch para poder realizar el bulk de datos
from elasticsearch import Elasticsearch, helpers
#Importamos el Datetime
from datetime import datetime
#Importamos Time para medir el tiempo que tarda en subirse
import time
#Empezamos a contar el tiempo
inicio = (time.time())
# Declaramos la instancia de Elasticsearch para poder conectarnos a
ella
cliente = Elasticsearch("localhost:9200")
# Función de Carga de fichero de texto
def get_data_de_fichero(self):
    # Devuleve lista de documentos
    return [l.strip() for l in open(str(self), encoding="utf8",
errors='ignore')]
# Recuperamos los datos
docs = get_data_de_fichero("TestElasticsearch.json")
# Imprimir longitud del documento
print ("Longitud de documento:", len(docs))
# define an empty list for the Elasticsearch docs
lista_docs = []
# use Python's enumerate() function to iterate over list of doc
strings
for num, doc in enumerate(docs):
    # catch any JSON loads() errors
    try:
        # convert the string to a dict object
        dict_doc = json.loads(doc)
        #Añadir un campo TimeStamp Actual de la fecha de subida
        dict_doc["timestamp"] = datetime.now()
        # append the dict object to the list []
        lista_docs += [dict_doc]
    except json.decoder.JSONDecodeError as err:
        # print the errors
        print ("ERROR para num:", num, "-- JSONDecodeError:", err,
"for doc:", doc)
print ("Longitud lista de documentos:", len(lista_docs))
try:
    print ("\nIndexación de lista documentos")
    # Usar el método bulk para indexar los documentos
    respuesta = helpers.bulk(cliente, lista_docs, index = "tfg-
json-elastic", doc_type = "TestElasticsearch.json")
    # print the response returned by Elasticsearch
    print ("Respuesta helpers.bulk():", respuesta)
    print ("Respuesta helpers.bulk():", json.dumps(respuesta,
indent=4))
    print("Tiempo para subir de Json a Elastic")
    fin = (time.time())
    print(fin-inicio)
except Exception as err:
    print("Elasticsearch helpers.bulk() ERROR:", err)
```

```
quit()
```

D Manual del Programador

El manual de Programador para este TFG consiste mayoritariamente en Docker y en Webservices.

Docker:

Comando para ver los contenedores activos:

$ docker ps

Comando para ver todos los contenedores:

$ docker ps -a

Comando para detener los contenedores:

$ docker stop <id contenedor>

Comando para borrar contenedores:

$ docker rm <id contenedor>

Para ejecutar un Docker-Compose.yml:

$ docker-compose up -d

Webservices:

Comando para iniciar un webservice:

$ sudo systemctl start <webservice>

Comando para comprobar el estado de un webservice:

$ sudo systemctl status <webservice>

Comando para detener el Servicio:

$ sudo systemctl stop <webservice>

Comando para reiniciar el Servicio:

$ sudo systemctl restart <webservice>

Logstash:

Para ejecutar Logstash es necesario estar en su carpeta bin. Para encontrar la dirección de Logstash:

$ sudo whereis Logstash.

Para ejecutar un Logstash.conf:

$ ./logstash -f logstash.conf